J Sched (2012) 15:323–332
DOI 10.1007/s10951-011-0225-1

Scheduling of pipelined operator graphs

Hans L. Bodlaender · Petra Schuurman · Gerhard J. Woeginger

Published online: 18 February 2011

Abstract We investigate a class of scheduling problems that arise in the optimization of SQL queries for parallel machines (Chekuri et al. in PODS'95, pp. 255–265, 1995). In these problems, an undirected graph is used to represent communication and inter-operator parallelism. The goal is to minimize the global response time of the system.

We provide a polynomial time approximation scheme for the special cases where the operator graph is a tree, thereby improving on a polynomial time 2.87-approximation algorithm by Chekuri et al. The approximation scheme is generalized to the case where the operator graph has treewidth bounded by a constant. We analyze instances with small response times for general operator graphs: Deciding whether a response time of four time units can be reached is easy, but deciding whether a response time of six time units can be reached is NP-hard. Finally, we prove that for general operator graphs the existence of a polynomial time approximation algorithm with worst case performance guarantee better than $4/3$ would imply P = NP.

Keywords Query optimization · SQL · Operator graph · Pipelined parallelism · Scheduling · Worst case analysis

1 Introduction

Chekuri et al. (1995) study scheduling problems that arises in the optimization of SQL queries for parallel machines. Their work is motivated by Gray (1988) who demonstrates that communication is a significant component of the cost of processing an query in parallel. An *operator graph* $G = (V, E)$ is an undirected graph with (positive integer) weights on the vertices and edges. The vertices in $V = \{1, 2, \dots, n\}$ are called *operators* and represent atomic units of execution. The (positive integer) weight t_i of operator i represents the cost of executing this operator. The (positive integer) weight c_{ij} of an edge $[i, j] \in E$ represents the remote communication between the two incident operators. Moreover, there are p processors available on which the operators are to be processed.

A *schedule* assigns operators to processors. More precisely, a schedule partitions V into p sets $V_1, \dots, V_p$ where set V_k is allocated to the kth processor. Since edge weights represent the cost of communication, this cost is saved whenever two adjacent operators share a processor. With this, the *load* L_k on the kth processor is the cost for executing all operators in V_k plus the overhead for communicating with other processors, that is,

$$L_k = \sum_{i \in V_k} t_i + \sum_{i \in V_k} \sum_{j \notin V_k} c_{ij}. \qquad (1)$$

This research has been supported by the Netherlands Organization for Scientific Research (NWO), grant 639.033.403; by DIAMANT (an NWO mathematics cluster); and by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society). A preliminary version of this paper has appeared as an extended abstract in the proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California: P. Schuurman and G.J. Woeginger, Scheduling a pipelined operator graph, pp. 207–212.

H.L. Bodlaender
Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

P. Schuurman · G.J. Woeginger (✉)
Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: gwoegi@win.tue.nl

The *response time* (or *makespan*) of a schedule is the maximum processor load $L = \max_{1 \le k \le p} L_k$. The goal is to find a schedule that minimizes the response time. The smallest possible response time is denoted by L^*.

Known and related results Scheduling of operator graphs is an NP-hard problem, even in case there is no communication between operators and the graph is just a collection of independent vertices: This special case is the classical NP-hard makespan minimization on parallel identical processors; cf. Garey and Johnson (1979). Hasan and Motwani (1994) developed a polynomial time approximation algorithm with constant worst case guarantees for the special case where the operator graph is a path or a star. Chekuri et al. (1995) then studied the special case were the operator graph is a tree, and derive a polynomial time 2.87-approximation algorithm.

A closely related scheduling model is scheduling under communication delays; see for instance the survey article by Veltman et al. (1990). In this model, the operators/tasks are precedence constrained by some partial order, and every precedence constraint $i \to j$ comes with a communication delay $cd(i, j)$. If a schedule assigns tasks i and j to the same processor, then the earliest possible start time of j is simply the completion time of i. If the schedule assigns i and j to different processors, then the earliest possible start time of j is the completion time of i plus the corresponding communication delay $cd(i, j)$. A common objective in scheduling under communication delays is to minimize the maximum task completion time.

The main differences between the operator graph model and the communication delay model are as follows. First, a schedule in the communication delay model specifies both, the assignment of tasks to processors and the time slots during which the individual tasks are processed. In the operator graph model, a schedule only specifies the assignment of tasks to processors. Secondly, in the communication delay model every processor can communicate with other processors and simultaneously process its tasks. In the operator graph model, processors are idle during communications and hence the communication times contribute to the work loads of the processors.

Results of this paper In the current paper we extend and improve the results of Chekuri et al. (1995) for various natural classes of operator graphs. Our main results are the following.

- First, we illustrate the existence of a polynomial time approximation scheme (PTAS) for the case where the operator graph is a tree, that is, for the special case that has been considered by Chekuri et al. (1995).
- This scheme is then generalized to graphs of bounded treewidth, that is, we show that there is a PTAS for the case that the operator graph has treewidth bounded by a constant k.
- We prove that for general operator graphs deciding whether a response time of 6 time units can be reached is NP-hard, whereas deciding whether a response time of 4 time units can be reached is easy. (The case of 5 time units remains open.)
- Finally, we deduce from the NP-hardness result that the existence of a polynomial time approximation algorithm with worst case performance guarantee better than $4/3$ implies P = NP.

The paper is organized into sections as follows. In Sects. 2 and 3 we derive the PTAS for trees: Sect. 2 gives an approximate dynamic programming formulation for this problem that has pseudo-polynomial running time while only introducing a reasonably small (and controllable) error. Section 3 brings the running time down to polynomial and pays for this with another (small and controllable) error factor. Section 4 gives the polynomial time approximation scheme for graphs of bounded treewidth. Then we turn to the case of schedules with small response times: Sect. 5 proves the polynomial time result for response time four, and Sect. 6 proves the NP-hardness result for response time six. Section 7 deduces the in-approximability result of 4/3, and Sect. 8 gives the conclusions.

2 Approximate dynamic programming for operator trees

In this section we present an approximate dynamic programming formulation for minimizing response time on p processors for operator trees. The formulation is approximate in so far that it does not yield the exact optimal response time, but only a $(1 + \varepsilon)$-approximation of it.

Throughout this section, the operator graph $G = (V, E)$ is a tree with n vertices that are to be processed on p processors. Let D be the makespan of the schedule constructed according to the approximation algorithm of Chekuri et al. (1995). Hence,

$$L^* \le D \le 3L^*. \tag{2}$$

Let $\varepsilon > 0$ be an arbitrarily small positive real, and fix an integer d that satisfies $1/d \le \varepsilon$.

We root the tree $G = (V, E)$ at an arbitrary vertex $r \in V$. For each operator $v \in V$ we fix an arbitrary left-to-right ordering of its sons. For a vertex i, the *maximal subtree* rooted at i is the subtree that consists of vertex i together with all its descendants. With every edge $[i, j] \in E$ where j is the father of i, we associate a subtree $T(i, j)$ in the following way: The tree $T(i, j)$ consists of vertex j, of the maximal subtree rooted at vertex i, and of all the maximal subtrees rooted at left brothers of vertex i.

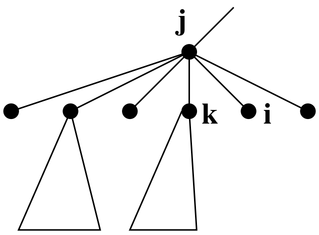

Fig. 1 An illustration for the handling of $T(i, j)$ in Case 1

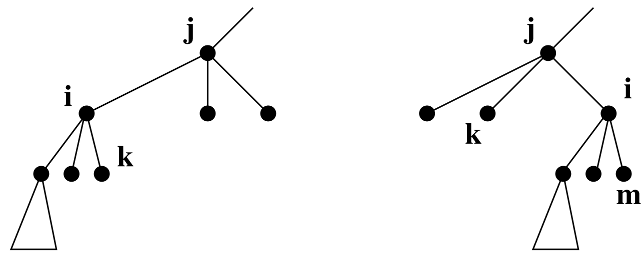

Fig. 2 An illustration for the handling of $T(i, j)$ in Case 2 and Case 3

Now consider a tree $T(i, j)$ for some fixed edge $[i, j]$. In any schedule, some of the edges in $T(i, j)$ will be broken since their endpoints go on different processors; the remaining (unbroken) edges will glue some of the vertices together. Hence, the schedule decomposes the tree $T(i, j)$ into connected components. The *cost* of a connected component $W \subseteq V$ in $T(i, j)$ is the sum of all weights t_h with $h \in W$ plus the sum of all communication costs c_{hk} with $h \in W$ and $k \in T(i, j) - W$. We stress that this cost is defined with respect to the subtree $T(i, j)$, and that it does not cover the cost of the communication for the edge leaving the root of $T(i, j)$. The connected component that contains the root j of $T(i, j)$ is called an *open* component, since it may be extended to also contain the father of j together with other vertices. All other components are called *closed* components. With every such partition of $T(i, j)$ into connected components, we associate a non-negative integer vector with $d^2 - d + 2$ components:

$$(A, B, n_{d+1}, n_{d+2}, \dots, n_{d^2}). \tag{3}$$

The meaning of this vector is the following: (i) The cost of the open component equals A. (ii) The total sum of the costs of all closed components with cost $\leq D/d$ equals B. (iii) For $\ell = d+1, \dots, d^2$, there are precisely n_ℓ closed components in the partition whose cost is greater than $(\ell - 1)D/d^2$ and less or equal to $\ell D/d^2$.

In the dynamic programming formulation, we will compute for every edge $[i, j]$ a set $\mathcal{S}(i, j)$ that contains all possible $(d^2 - d + 2)$-dimensional vectors that can be associated with a partition of $T(i, j)$ into connected components as described above. These computations are done bottom-up and from left-to-right. Hence, when we treat edge $[i, j]$, all edges below $[i, j]$ in G have already been treated and all edges from the father j to a left brother of i have already been treated. For trees $T(i, j)$ that consist of the single edge $[i, j]$, these computations are straightforward to do. Otherwise, $T(i, j)$ consists of at least two edges and we distinguish several cases; see Figs. 1 and 2 for an illustration.

- (Case 1). i is a leaf. Since $T(i, j)$ consists of at least two edges, i has a left brother vertex k. For every vector in $\mathcal{S}(k, j)$ we may either add vertex i to the open component (add t_i to the entry A) or we make i to form its own component (add c_{ij} to the entry A and update the information on the closed components appropriately).
- (Case 2). i is the leftmost son of j. Since $T(i, j)$ consists of at least two edges, i is not a leaf. Let k denote the rightmost son of i, such that $T(k, i)$ is the maximal subtree rooted at i. For every vector in $\mathcal{S}(k, i)$ we may either add vertex j to the open component (add t_j to the entry A) or we make j to form the new open component (there is a new closed component of cost $A + c_{ij}$; update the information on the closed components appropriately; set the new value of A to $t_j + c_{ij}$).
- (Case 3). i is neither a leaf nor the leftmost son of j. Denote by k the left brother of i, and denote by m the rightmost son of i. We have to combine all vectors in $\mathcal{S}(k, j)$ with all vectors in $\mathcal{S}(m, i)$. Either the open component in $T(m, i)$ is closed and the open component in $T(k, j)$ survives and becomes the new open component, or the two open components are glued together into a single new open component. In both cases, the information in both vectors is easily updated and merged into a single new vector.

The last edge treated will be the edge $[i, r]$ that connects the root r to its rightmost son i. In the very end, we close in every vector in $\mathcal{S}(i, r)$ the open component and translate it into a vector $(B, n_{d+1}, \dots, n_{d^2})$ with only $d^2 - d + 1$ components. Denote by $\mathcal{S}^*$ the resulting set of vectors.

Now the vectors in $\mathcal{S}^*$ essentially encode instances of the classical makespan minimization problem on parallel identical processors: Every connected component must go together onto a common processor. Communication along edges has become irrelevant, since the communication costs have been merged into the costs of the connected components. In every such instance, we have p processors, we have many small jobs whose total processing time is B, and for $\ell = d+1, \dots, d^2$ we have n_ℓ jobs of size $\ell D/d^2$. Note that we do *not* distinguish between the different values in the interval from $(\ell - 1)D/d^2$ to $\ell D/d^2$; the resulting error is at most a multiplicative factor of $(d+1)/d = 1 + \varepsilon$ in the response time. In fact, we end up in a situation as described by Hochbaum and Shmoys (1987) or Alon et al. (1998). By directly applying their machinery, the described instances can be formulated as integer linear programs whose dimension is a fixed constant (whose value only depends on ε and d). Then Lenstra's algorithm (Lenstra 1983) yields a polynomial time solution for the makespan problem. We solve all these makespan problems in polynomial time (per problem) and find the vector that leads to the best makespan. This vector is then translated back into a schedule for the operators. The resulting response time is at most a factor of $1+\varepsilon$ above the best possible response time.

What about the running time of this approach? The running time mainly depends on the cardinalities of the vector sets $\mathcal{S}(i,j)$ and $\mathcal{S}^*$. The last d^2-d entries of these vectors can only take $O(n^{d^2-d})$ different values; this number is polynomial in the input size, since d is a fixed constant. However, the first two entries of the vectors may take any value between 0 and $\sum t_i + \sum c_{ij}$; this number is only pseudo-polynomial in the input-size. Hence, the cardinalities of these vector sets are pseudo-polynomial in the input-size and so is the running time of the whole approach.

Theorem 2.1 *For every* $\varepsilon > 0$, *there exists a pseudo-polynomial time approximation algorithm that computes a* $(1+\varepsilon)$*-approximation of the best response time for operator trees.*

3 A PTAS for operator trees

In this section, we round the dynamic programming formulation of the preceding section such that its running time becomes polynomial. In doing this, we lose another factor of $1+\varepsilon$ in the response time. All in all, this gives the PTAS.

The main idea is the following: As soon as we have computed the vector set $\mathcal{S}(i,j)$, we look for a pair of vectors that are 'close' to each other. Two vectors $(A, B, n_{d+1}, \ldots, n_{d^2})$ and $(A', B', n'_{d+1}, \ldots, n'_{d^2})$ are called *close to each other*, if they agree in their last d^2-d components and if

$$\frac{1}{\Delta}A \le A' \le \Delta A \quad \text{and} \quad \frac{1}{\Delta}B \le B' \le \Delta B \tag{4}$$

holds, where $\Delta = 1+\varepsilon/(2n)$. As long as $\mathcal{S}(i,j)$ contains such a pair of close vectors, we remove one of these vectors from $\mathcal{S}(i,j)$. We denote the resulting cleaned-up version of $\mathcal{S}(i,j)$ by $\mathcal{S}'(i,j)$. All further computations are then done with $\mathcal{S}'(i,j)$. This technique is of trimming the state space, is due to Ibarra and Kim (1975); see also Woeginger (2000).

Lemma 3.1 *The cardinality of every cleaned-up set* $\mathcal{S}'(i,j)$ *is polynomial in the input-size.*

Proof In any vector in $\mathcal{S}(i,j)$, the entries A and B are integers in the range from 0 to $X := \sum t_i + \sum c_{ij}$. The remaining d^2-d entries are integers in the range 0 to n. By (4), the cleaned-up version contains at most

$$\begin{aligned}\log^2_\Delta(X)\cdot n^{d^2-d} &= \ln^2(X)/\ln^2(\Delta)\cdot n^{d^2-d}\\ &\le (1+2n/\varepsilon)^2\ln^2(X)\cdot n^{d^2-d}\end{aligned}$$

vectors; here we have used that $\ln x \ge (x-1)/x$ for $x \ge 1$. Clearly, this bound is polynomial in n and in the input-size $\ln(X)$. □

We conclude that the cleaning-up indeed brings the running time down to polynomial. What about the error that is introduced by the cleaning-up? Every time we clean up a vector set $\mathcal{S}(i,j)$, we introduce a new multiplicative error of Δ. All together, there are at most n cleaning steps, and thus the total error introduced by cleaning is at most

$$\Delta^n = \left(1+\frac{\varepsilon}{2n}\right)^n \le 1+\varepsilon.$$

Theorem 3.2 *The problem of minimizing response time for operator trees on identical processors possesses a PTAS.*

4 Operator graphs that have bounded treewidth

Many combinatorially hard NP-complete problems are known to be polynomial time solvable on graphs of bounded treewidth. Such results usually generalize similar polynomial time solvability results for trees; see for instance Bodlaender (1997). In this section, we show how to extend the results of the previous two sections to graphs of bounded treewidth.

Graphs of treewidth at most k, also known as partial k-trees, can be defined in several equivalent ways. We use here a definition, based on the work of Borie et al. (1991) and Wimer (1987), which is most useful for our purposes; see also Bodlaender (1998). Throughout this section, we will assume that k is a constant.

Definition 4.1 A *terminal graph* is a triple (V, E, X), where V is a set of vertices, E is a set of edges (unordered pairs of vertices from V), and X is an ordered subset of V, called the set of *terminals*. If $|X| = k$, then (V, E, X) is called a k-terminal graph.

Now, a graph $G = (V, E)$ has treewidth at most k, if and only if terminal graph $(V, E, \emptyset)$ can be constructed with help of the following four operations (Bodlaender 1998; Wimer 1987):

1. **Leaf**. Take a terminal graph with one vertex, which is a terminal, and no edges.
2. **Join**. Take two terminal graphs $G_1 = (V_1, E_1, X_1)$, $G_2 = (V_2, E_2, X_2)$ with the same number of terminals. Take the disjoint union of G_1 and G_2, and then identify the ith terminal of G_1 with the ith terminal of G_2, for all i, $1 \le i \le |X_1| = |X_2|$.
3. **Introduce**. Take a terminal graph $G = (V, E, X)$ with $|X| \le k$. Add a new vertex v, which is also a terminal,

and zero or more edges from v to terminals in X, that is, take a new terminal graph $(V \cup \{v\}, E \cup \{\{v, y\} | y \in Y\}, X \cup \{v\})$, with $Y \subseteq X$.

4. **Forget**. Take a terminal graph $G = (V, E, X)$, and turn one terminal vertex into a non-terminal vertex, that is, take $(V, E, X - \{v\})$ for $v \in X$.

If one has a graph $G = (V, E)$ of treewidth at most k, then one can find a tree decomposition of G of width at most k in linear time (Bodlaender 1996). This tree decomposition can then be transformed, again in linear time, to a tree (the *construction tree*) with nodes labeled by Leaf, Join, Introduce, or Forget operations, such that the tree represents the construction of $G = (V, E, \emptyset)$ as terminal graph. We now suppose we have this construction tree.

Note that to each node of the construction tree, we have an associated terminal graph. We will often not distinguish between a node in the construction tree and this associated terminal graph. As we do only Introduce operations to terminal graphs with at most k terminals, we never deal with a terminal graph with more than $k + 1$ terminals.

Let $\varepsilon > 0$ be a positive real number, let d be an integer such that $1/d < \varepsilon$, and let $D = \sum_{v \in V} t_v + \sum_{\{v,w\} \in E} c_{vw}$.

Consider a terminal graph (V', E', X). A *proper partition* of V' is a partition $(V_1, \dots, V_r)$ of V' in one or more disjoint sets, such that

- Each set V_i that does not contain a terminal vertex ($V_i \cap X = \emptyset$) induces a connected sub-graph.
- If set V_i contains a terminal vertex, then every connected component of the sub-graph induced by V_i contains a terminal vertex.
- $\bigcup_{1 \le i \le r} V_i = V'$.

If a set V_i does not contain a terminal vertex, we call V_i a closed set; otherwise we call V_i an open set. The *cost* of a set V_i is defined as before, ignoring edges to vertices not in V'; that is, the cost of V_i with respect to (V', E', X) is

$$\sum_{v \in V_i} t_i + \sum_{v \in V_i, w \notin V_i, \{v,w\} \in E'} c_{vw}.$$

To a terminal graph $G' = (V', E', X)$ and a proper partition $(V_1, \dots, V_r)$ of V', we associate the *tuple* of the partition and G':

$$(\simeq, \simeq_0, f, B, n_{d+1}, \dots, n_{d^2}).$$

Here, $\simeq$ is an equivalence relation on the set of terminals X, with for all $v, w \in X$, $v \simeq x$, if and only if v and w belong to the same set V_i, $1 \le i \le r$, and $\simeq_0$ is an equivalence relation on the set of terminals X, with for all $v, w \in X$, $v \simeq_0 x$, if and only if $v \simeq w$ and v and w belong to the same connected component of the sub-graph induced by the set V_i with $v, w \in V_i$. Furthermore, $f : X \to \mathbf{N}$ maps each terminal vertex v to the cost of the open set to which v belongs, B is the total cost of all closed sets that have cost at most D/d, and for $\ell = d + 1, \dots, d^2$, there are exactly n_ℓ closed components whose cost is greater than $(\ell - 1)D/d^2$ and less or equal to $\ell D/d^2$.

A set of tuples of all possible proper partitions for a given terminal graph $G' = (V', E', X)$ is called the *active set*. The idea is to compute an active set for every (graph represented by a) node in the construction tree of G. This is done bottom-up in the construction tree: an active set of tuples for a node in the construction tree can be computed from the active sets of the children of the node. We now discuss for each of the four types of nodes how this computation can be done.

Leaf nodes The terminal graph associated with a leaf node is of the form $(\{v\}, \emptyset, \{v\})$, and the one possible proper partition gives the following tuple: $(\simeq, \simeq, f, 0, 0, \dots, 0)$, where $f(v) = t_v$, and $\simeq$ is the trivial equivalence relation on the one-element set $\{v\}$. The active set for this leaf node consists of this tuple.

Join nodes Suppose $G^0 = (V^0, E^0, X)$ is formed by a join of $G^1 = (V^1, E^1, X)$ and $G^2 = (V^2, E^2, X)$. (To ease presentation, we use the same name for a vertex that is identified with another vertex as for this other vertex and for the resulting vertex. So, each of these three graphs has the same set of terminals.)

Suppose active sets for G^1 and G^2, $\mathcal{A}_1$, $\mathcal{A}_2$ are known. To compute an active set of tuples for G^0, we take a set $\mathcal{A}$ that is initially empty.

We now consider each pair of tuples $(\simeq, \simeq_0, f, B, n_{d+1}, \dots, n_{d^2})$ from the active set of G^1 and $(\simeq, \simeq_0', f', B', n'_{d+1}, \dots, n'_{d^2})$ from the active set of G^2. Note that we only consider pairs with the first equivalence relation identical. The idea is that we build a new proper partition from the proper partition, represented by these tuples, in the following way: every closed set in one of these proper partitions is again a closed set in the new proper partition of G^0. Every open set in the new proper partition is the union of an open set in G^1 and an open set G^2 with the same terminals. The tuple

$$(\simeq, \simeq_0'', f'', B'', n''_{d+1}, \dots, n''_{d^2})$$

for this new proper partition can be computed from the two given tuples, in the following way.

Compute the equivalence relation $\simeq_0''$, which is the transitive closure of the relation R with $vRw \Leftrightarrow v \simeq_0 w \vee v \simeq_0' w$. This new equivalence relation precisely captures when v and w belong to the same connected component of $G^0[V_i]$ for some V_i, as v and w belong to the same component in G^0 when they do so in G^1 or in G^2 or via intermediate vertices in X that belong to the same component: this is precisely captured by taking the transitive closure.

For $v \in X$, the value $f''(v)$ can be computed as follows. The cost of the set to which v belongs basically is the cost of the part of the set that belongs to G^1 plus the cost of the part of the set that belongs to G^2, but we have to subtract those costs we counted twice. Thus, we obtain

$$f''(v) = f(v) + f'(v) - \sum_{w \in S(v)} t_w - \sum_{w \in S(v), \{v,w\} \in E^0} c_{vw}.$$

Here, $S(v) = \{w \in X | v \simeq w\}$ is the set of terminals that belong to the same open set as v in the proper partition.

Furthermore, as the new collection of closed components is just the disjoint union of the collections of closed components for G^1 and for G^2, one can directly see that the proper values for B'', and n''_ℓ are obtained by taking $B'' = B + B'$ and $n''_\ell = n_\ell + n'_\ell$, $d + 1 \leq \ell \leq d^2$. Add the tuple thus computed to the set $\mathcal{A}$. After all pairs are dealt with, $\mathcal{A}$ forms an active set of tuples for G^0.

Introduce nodes Suppose $G^0 = (V^0, E^0, X \cup \{v\})$ is obtained from $G^1 = (V^1, E^1, X)$ by an introduce operation; v the newly introduced vertex. Suppose $Y = \{w \in X | \{v, w\} \in E^0\}$.

Again, we build an active set $\mathcal{A}$ for G^0 from an active set $\mathcal{A}'$ for G^1. $\mathcal{A}$ is initially empty.

For each tuple $(\simeq, \simeq_0, f, B, n_{d+1}, \dots, n_{d^2})$ from $\mathcal{A}'$, we add two or more tuples to $\mathcal{A}$, representing the different cases where v can be added in the partition. The idea is that we use a proper partition, represented by this tuple, and either add one additional set $\{v\}$, or add v to one of the (necessarily open) sets in the partition.

First, we look to the case that v forms a set on its own. Let $\simeq'$ be the equivalence relation on $X \cup \{v\}$, with $x \simeq' y$ iff $x, y \in X$ and $x \simeq y$, or $x = y = v$. Similarly, let $\simeq'_0$ be the equivalence relation on $X \cup \{v\}$, with $x \simeq'_0 y$ iff $x, y \in X$ and $x \simeq_0 y$, or $x = y = v$. Take $f'(w) = f(w) + \sum_{w \simeq x, \{v,x\} \in E^0} c_{vw}$ for $w \in X$, and $f'(v) = t_v + \sum_{w \in Y} c_{vw}$. Note that f' gives for each w the cost of the set containing w in the new proper partition of G^0. Thus, this case corresponds to tuple $(\simeq', \simeq'_0, f', B, n_{d+1}, \dots, n_{d^2})$; this tuple is added to $\mathcal{A}$.

Then, for every $w \in X$, we consider the case that v is added to the set that contains w. Here, $\simeq'$ is the equivalence relation on $X \cup \{v\}$, with $x \simeq' y$ iff $x, y \in X$ or if $v = x$ and $w \simeq y$. $\simeq'_0$ is the transitive closure of the relation R on $X \cup \{v\}$, with xRy, iff $x, y \in X$ and $x \simeq_0 y$ or $x = v$ and $y \in Y$, or $y = v$ and $x \in Y$. One can see that now for all $x, y \in X \cup \{v\}$, $x \simeq'_0 y$, if and only if x and y belong to the same connected component in the newly formed proper partition (i.e., adding v to the set of w.) The costs are as follows. For $x \in X$ with not $x \simeq w$, $f'(x) = f(x) + \sum_{y \simeq x, y \in Y} c_{vy}$. For $x = v$ or $x \in X, x \simeq w$, $f'(x) = f(w) + \sum_{y \not\simeq w, y \in Y} c_{yv}$. Now, the tuple $(\simeq', \simeq'_0, f', B, n_{d+1}, \dots, n_{d^2})$ corresponds to the proper partition where v is added to the same set as w; this tuple is added to $\mathcal{A}$.

The resulting set $\mathcal{A}$ is an active set of G^0.

Forget nodes Suppose $G^0 = (V^0, E^0, X)$ is obtained by a forget operation from $G^1 = (V^0, E^0, X \cup \{v\})$. An active set of G^0 can be obtained by an active set by a kind of 'projection'.

Again, take an empty set $\mathcal{A}$. For every tuple $t = (\simeq, \simeq_0, f, B, n_{d+1}, \dots, n_{d^2})$ from the active set of G^1, do the following. One of the following cases must hold.

1. There is a $w \in X$ with $v \simeq w$, but there is no $y \in X$ with $v \simeq_0 y$. In this case, the corresponding partition is no longer proper: the connected component containing v belongs to a larger set, but no longer contains a terminal vertex. Here, we just ignore the tuple.
2. There is a $w \in X$ with $v \simeq_0 w$. The proper partition remains proper, and has the same closed and open sets. Let $\simeq'$, $\simeq'_0$, f' be the restrictions to X of $\simeq$, $\simeq_0$, f, respectively. Then, $(\simeq', \simeq'_0, f', B, n_{d+1}, \dots, n_{d^2})$ is the tuple representing the same proper partition as the tuple t, but now for G^0 instead of for G^1. Add this tuple to $\mathcal{A}$.
3. There is no $w \in X$ with $v \simeq_0 w$. In this case, the proper partition remains proper, but the open set that contained v now becomes closed. Let again be $\simeq'$, $\simeq'_0$, f' the restrictions to X of $\simeq$, $\simeq_0$, f, respectively. There are two subcases.
   (a) If $f(v) \leq D/d$, then the cost of the newly formed closed components must be added to B, and hence we insert the tuple $(\simeq', \simeq'_0, f', B + f(v), n_{d+1}, \dots, n_{d^2})$ in $\mathcal{A}$.
   (b) If $(\ell - 1)D/d^2 < f(v) \leq \ell D/d^2$ $(d + 1 \leq \ell \leq d^2)$, then insert the tuple $(\simeq', \simeq'_0, f', B, n_{d+1}, \dots, n_{\ell-1}, n_{ell} + 1, n_{\ell+1}, \dots, n_{d^2})$ in $\mathcal{A}$.

Again, the set $\mathcal{A}$ that results after all tuples in the active set of G^1 are considered is an active set of G^1.

Note that each active set has a number of elements that is pseudo polynomial, assuming that the number of terminals of the terminal graphs we deal with is bounded by a constant k. It follows that we can compute in polynomial time the active set of the root node of the construction tree.

Note that the terminal graph corresponding to this root node is $(V, E, \emptyset)$. As we have no terminals, we can ignore the first three elements of the tuples, and arrive at a set of vectors of the form $(B, n_{d+1}, \dots, n_{d^2})$. These vectors have the same form, and the same meaning as those used in the algorithm for trees. Thus, the same technique as used for trees can then be applied, and we arrive at the following generalization of Theorem 2.1.

Theorem 4.2 *For every $\varepsilon > 0$, and every $k \in \mathbf{N}$, there exists a pseudo-polynomial time approximation algorithm that* *computes a* $(1+\varepsilon)$*-approximation of the best response time for operator graphs of treewidth at most* k.

To obtain a PTAS for graphs of bounded treewidth, we generalize the clean-up method. Every time an active set is computed, a clean-up of the active set is done, as follows.

Two tuples $(\simeq, \simeq_0, f, B, n_{d+1}, \dots, n_{d^2})$ and $(\simeq', \simeq_0', f', B', n'_{d+1}, \dots, n'_{d^2})$ are called to be *close to each other*, when $\simeq=\simeq'$, $\simeq_0=\simeq_0'$, $n_\ell = n'_\ell$, $d+1 \le \ell \le n^2$, for all terminals v, $f(v)/\Delta \le f'(v) \le \Delta f(v)$, and $B/\Delta \le B' \le \Delta B$, where $\Delta = 1 + \varepsilon/(2n)$, where n is the number of nodes in the construction tree. As long as there are two tuples that are close to each other in the active set, one of them is removed in the clean-up operation.

The remaining analysis is identical to the case of trees, and we obtain the following result:

Theorem 4.3 *For every fixed* k, *the problem of minimizing response time for operator graphs of treewidth at most* k *on identical processors possesses a PTAS.*

5 A polynomial time result for response time four

In this section, we prove that for arbitrary operator graphs, we can decide in polynomial time whether there exists a schedule with response time ≤ 4 on p processors. Let us start with several simple observations that every YES-instance with response time ≤ 4 must satisfy. Clearly, every node i in such an instance has execution cost $1 \le t_i \le 4$.

Lemma 5.1 *If there is a node* i *of degree at least* 4, *then the instance is a NO-instance.*

Proof Consider an arbitrary schedule, and consider the workload L_k of the processor that is processing i. Then the execution cost of i contributes at least 1 to L_k, and each of the 4 neighbors contributes a communication cost or execution cost of at least 1 to L_k. Hence $L_k \ge 5$, and the instance is a NO-instance. □

Lemma 5.2 *If there is a node* i *of degree* 3, *then the considered instance can be reduced* (*in polynomial time*) *to an equivalent smaller instance with fewer processors.*

Proof Let x_1, x_2, x_3 be the three neighbors of node i. Consider an arbitrary schedule, and consider the workload L_k of the processor that is processing i. By arguing as in the preceding lemma, we see that the only possibility for a YES-instance is that node i has execution cost 1, and that every neighbor x_j $(1 \le j \le 3)$ contributes 1 to the workload L_k either through its execution cost or through its communication cost.

We initialize the set $S := \{i, x_1, x_2, x_3\}$ of operators scheduled together with node i, and then perform the following two removal steps until the situation stabilizes:

- If x_j has execution cost at least 2, then remove x_j from S.
- If x_j has a neighbor outside of S, then remove x_j from S.

If the resulting set S contains node x_j, then x_j has execution cost 1 and incurrs no communication costs to nodes outside of S. Hence we may assign the nodes in S to a common processor.

We remove the nodes in S from the instance, we decrease the number of processors by one, and we increase the execution times of the surviving neighbor x_j so that they cover the communication costs with the nodes in S. This clearly yields an equivalent instance with fewer processors. □

Lemmas 5.1 and 5.2 leave us with an operator graph in which all nodes are of degree at most 2. The connected components of such a graph are isolated nodes, paths, and cycles.

We now work through these connected components one by one. Hence consider some fixed component C, and consider some fixed four-tuple (n_1, n_2, n_3, n_4) with $n_1 + n_2 + n_3 + n_4 \le p$. We define a Boolean variable $X[C; n_1, n_2, n_3, n_4]$ with the following meaning: "*Component* C *can be cut into* $n_1 + n_2 + n_3 + n_4$ *connected pieces such that for* $r = 1, 2, 3, 4$ *there are exactly* n_r *pieces with load* r." Since C is a path or a cycle, $X[C; n_1, n_2, n_3, n_4]$ can easily be computed in polynomial time by straightforward dynamic programming (moving along the path or moving along the cycle). Since there are only $O(p^4)$ four-tuples (n_1, n_2, n_3, n_4) with $n_1 + n_2 + n_3 + n_4 \le p$, the value of all Boolean variables can be determined in polynomial time.

In the final phase, we consider an enumeration $C_1, C_2, \dots, C_s$ of all connected components. For every four-tuple (n_1, n_2, n_3, n_4) with $n_1 + n_2 + n_3 + n_4 \le p$ and for every integer t with $1 \le t \le s$, we define a Boolean variable $Y[t; n_1, n_2, n_3, n_4]$ with the following meaning: "*There exists a feasible schedule for the operators in the first* t *components* $C_1, C_2, \dots, C_t$ *that for* $r = 1, 2, 3, 4$ *uses exactly* n_r *processors with load* r." Again, this is a standard optimization problem that can be solved by a standard dynamic programming approach based on the values $X[C; n_1, n_2, n_3, n_4]$.

In the end, we only have to check whether at least one of the Boolean variables $Y[s; n_1, n_2, n_3, n_4]$ has been set to true. This then yields a schedule of all nodes in all s components on at most $n_1 + n_2 + n_3 + n_4 \le p$ processors each with load at most 4.

Theorem 5.3 *For arbitrary operator graphs, it can be decided in polynomial time whether there exists a schedule with response time at most four.*

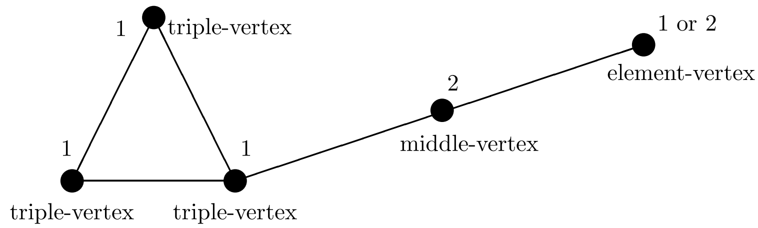

Fig. 3 Triple-vertices, middle-vertices, and element-vertices

6 A hardness result for response time six

In this section we investigate the general operator scheduling problem where the communication constraints are not restricted to be trees but may form an arbitrary graph $G = (V, E)$ on the operator set V. We show that it is NP-hard to decide whether there exists a schedule with response time 6.

We start with an NP-hardness proof for operator scheduling in arbitrary graphs. The reduction is from the following NP-hard version of the *3-Dimensional Matching* problem with bounded occurrence of elements (3DM-B); cf. Garey and Johnson (1979).

3-DIMENSIONAL MATCHING (3DM-B)

Input: Three sets $A = \{a_1, a_2, \dots, a_q\}$, $B = \{b_1, b_2, \dots, b_q\}$ and $C = \{c_1, c_2, \dots, c_q\}$. A subset T of $A \times B \times C$ of cardinality s, such that any element of A, B and C occurs in exactly two or three triples in T. Note that $s \geq q$.
Question: Does there exist a subset T' of T with $|T'| = q$ that covers every element in $A \cup B \cup C$?

From an instance of 3DM-B, we construct an operator graph $G = (V, E)$ in the following way: For every triple $t \in T$, there is a corresponding triangle in G whose vertices are called triple-vertices and correspond to the occurrences of the three elements in t. For every element in $A \cup B \cup C$, there is a corresponding element-vertex in G. If an element occurs in some triple, then there is a path of two edges that connects the corresponding element-vertex to the triple-vertex that corresponds to the occurrence of this element in this triple; the central vertex in this path is called a middle-vertex; see Fig. 3 for an illustration. This completes the description of the underlying graph G.

Next, let us specify the execution times and communication times of the vertices in G: The execution time of every triple-vertex is 1. The execution time of every middle-vertex is 2. The execution time of every element-vertex is 1 if the element occurs in three triples of T, and it is 2 is the element occurs in exactly two triples. The communication costs of all edges in E are 1. Finally, the number p of processors is set to $3s + q$.

Lemma 6.1 *If the* 3DM-B *instance has answer* YES, *then there exists a schedule with response time* 6.

Proof Let T' be a set of q triples that cover all elements in $A \cup B \cup C$. The following steps (a)–(c) show how to allocate all vertices on $3s + q$ processors with response time 6 and thus prove the claim.

(a) For every triple $t \in T'$, we process its three triple-vertices together on one processor. This gives a processor load of 6 ($1 + 1 + 1$ for executing the three vertices plus $1 + 1 + 1$ for communicating to the three adjacent middle-vertices). This consumes q processors.

(b) For every triple $t \notin T'$, we process its three triple-vertices plus the three adjacent middle-vertices on three processors. Every such processor processes one triple-vertex plus the adjacent middle-vertex; again this gives a processor load of 6 ($1 + 2$ for executing the vertices plus $1 + 1 + 1$ for communicating to the adjacent vertices). This consumes $3(s - q)$ processors.

(c) Since every element is contained in exactly one triple in T', for every element-vertex all but one of the adjacent middle-vertices have been scheduled in step (b). The remaining adjacent middle-vertex is processed together with the element-vertex on one processor. If the corresponding element is contained in three triples in T, then the total execution time assigned to this processor is $1 + 2$, and the total communication time is $1 + 1 + 1$. If the corresponding element is contained in two triples in T, then the total execution time assigned to this processor is $2 + 2$, and the total communication time is $1 + 1$. Again, this gives a processor load of 6. This consumes $3q$ processors. □

Lemma 6.2 *If there exists a schedule with response time at most* 6, *then the* 3DM-B *instance has answer* YES.

Proof Consider a schedule with response time 6. It is easy to see that an element-vertex cannot be processed on the same processor with a triple-vertex, or together with another element-vertex, or together with two middle-vertices. Moreover, a triple-vertex can only be processed alone, or together with one or two adjacent triple-vertices from the same triangle, or together with an adjacent middle-vertex. We now restructure the schedule in two phases.

In the first phase, we consider a processor P that only processes a single element-vertex. We choose an arbitrary middle-vertex that is adjacent to the element-vertex in G, and move it to processor P. This does not increase the load on the processor that loses the middle-vertex, and it cannot make the load on P larger than 6. This step is repeated over and over again, until every element-vertex is processed together with some middle-vertex.

In the second phase, we consider a processor P that processes a triple-vertex, but no middle-vertex. We move the adjacent triple-vertices from the same triangle to processor P; this does not increase the loads of the processors who lose vertices and makes the load of processor P exactly 6. This step is repeated until every triple-vertex is processed together with an adjacent middle-vertex or together with the other two triple-vertices in the same triangle.

At the end of the second phase, there only remain four types of processors: The $3q$ processors of type I process an element-vertex together with a middle-vertex. There are x processors of type II that process three triple-vertices together. There are y processors of type III that process a single triple-vertex, a single middle-vertex, or a triple-vertex together with a middle-vertex. Now since there are $3(s-x)$ triple-vertices on processors of type III, $y \geq 3(s-x)$ holds, and since there must be $3s-3q$ middle-vertices on processors of type III, $y \geq 3s-3q$ holds. Since the total number of used processors $3q+x+y$ is at most $3s+q$, we derive the two inequalities $q \leq x$ and $x \leq q$ from this. Hence, $x=q$ and $y=3s-3q$ must hold true, and every processor of type III processes a triple-vertex together with an adjacent middle-vertex.

Consider those $3q$ triple-vertices that are scheduled on processors type II. The $3q$ middle-vertices that are adjacent to these triple-vertices are all processed together with an adjacent element-vertex on a processor of type I. Consequently, the q triples that correspond to these q triangles cover all $3q$ elements in $A \cup B \cup C$. □

Lemmas 6.1 and 6.2 together imply the following theorem.

Theorem 6.3 *For arbitrary operator graphs, it is* NP-*hard to decide whether there exists a schedule with response time at most six.*

7 The in-approximability result

This section deduces another negative result from the construction in the preceding section. We stress that the classical gap technique would only yield a lower bound of 7/6. Hence another small trick is needed, and this trick is provided in the following technical lemma.

Lemma 7.1 *In any feasible schedule for the operator graph constructed in Sect. 6, the load of any processor is an even integer.*

Proof For a vertex i in G, define the pseudo-weight $w(i)$ to be its execution time plus the number of its incident edges. Note that for every vertex i in G the pseudo-weight $w(i)$ is an even integer.

Now consider a processor k that processes a set V_k of vertices. We claim that the parity of the load L_k equals the parity of $\sum_{i \in V_k} w(i)$: If an edge contributes a value 1 to the load, then exactly one of its endpoints is in V_k, and the edge contributes 1 to $\sum_{i \in V_k} w(i)$. If an edge does not contribute to the load, then either both or none of its endpoints are in V_k. Then the edge either contributes 0 or 2 to the sum $\sum_{i \in V_k} w(i)$. This proves the claim. Since all pseudo-weights are even, it also proves the lemma. □

Now assume for the sake of contradiction that for some $\varepsilon > 0$ there exists a polynomial time approximation algorithm for minimizing the response time on arbitrary operator graphs with worst case guarantee $4/3 - \varepsilon$. We apply this approximation algorithm to the instance G constructed in the preceding section. In case the approximation algorithm returns a schedule of length 6 or less, we know from Lemma 6.2 that the instance of 3DM-B has answer YES. By Lemma 7.1, the approximation algorithm cannot return a schedule of length 7. Finally, if the approximation algorithm returns a schedule of length 8 or more, then we know that the optimal response time is at least $8/(4/3-\varepsilon) > 6$. Lemma 6.1 yields that in this case the answer to the instance of 3DM-B is NO. Summarizing, we find in polynomial time the answer to any instance of the NP-hard problem 3DM-B, and this implies $P = NP$.

Theorem 7.2 *Unless* $P =$ NP, *every polynomial time approximation algorithm for minimizing the response time on arbitrary operator graphs has a worst case performance guarantee of at least* 4/3.

8 Conclusions

We performed a complexity and approximability analysis for minimizing response time in scheduling a pipelined operator graphs. We derived approximation schemes for graphs of bounded treewidth, and we established in-approximability of the general problem. We showed that deciding the existence of a schedule with response time four is easy, whereas deciding response time six is hard. The case with response time five remains open.

The approximability of the problem in general graphs remains open. We are not aware of any non-trivial results for it.

Acknowledgements We thank the referees for a careful reading of the paper and for helpful comments that improved the presentation of the paper.

References

Alon, N., Azar, Y., Woeginger, G. J., & Yadid, T. (1998). Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, *1*, 55–66.

Bodlaender, H. L. (1996). A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, *25*, 1305–1317.

Bodlaender, H. L. (1997). Treewidth: Algorithmic techniques and results. In I. Privara & P. Ruzicka (Eds.), *LNCS: Vol. 1295*. *Proceedings of the 22nd international symposium on mathematical foundations of computer science (MFCS'1997)* (pp. 19–36). Berlin: Springer.

Bodlaender, H. L. (1998). A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, *209*, 1–45.

Borie, R. B., Parker, R. G., & Tovey, C. A. (1991). Deterministic decomposition of recursive graph classes. *SIAM Journal on Discrete Mathematics*, *4*, 481–501.

Chekuri, C., Hasan, W., & Motwani, R. (1995). Scheduling problems in parallel query optimization. In *Proceedings of the 14th ACM symposium on principles of database systems (PODS'95)* (pp. 255–265).

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: a guide to the theory of* NP*-completeness*. San Francisco: Freeman.

Gray, J. (1988). The cost of messages. In *Proceedings of the 7th ACM symposium on principles of distributed computing (PODC'88)* (pp. 1–7).

Hasan, W., & Motwani, R. (1994). Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proceedings of the 20th international conference on very large databases* (pp. 36–47).

Hochbaum, D. S., & Shmoys, D. B. (1987). Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the Association of Computing Machinery*, *34*, 144–162.

Ibarra, O., & Kim, C. E. (1975). Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the Association of Computing Machinery*, *22*, 463–468.

Lenstra, H. W. (1983). Integer programming with a fixed number of variables. *Mathematics of Operations Research*, *8*, 538–548.

Veltman, B., Lageweg, B. J., & Lenstra, J. K. (1990). Multiprocessor scheduling with communication delays. *Parallel Computing*, *16*, 173–182.

Wimer, T. V. (1987). *Linear algorithms on k-terminal graphs*. PhD thesis, Dept. of Computer Science, Clemson University.

Woeginger, G. J. (2000). When does a dynamic programming formulation guarantee the existence of an FPTAS? *INFORMS Journal on Computing*, *12*, 57–75.